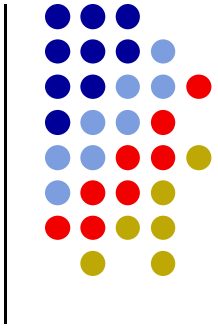
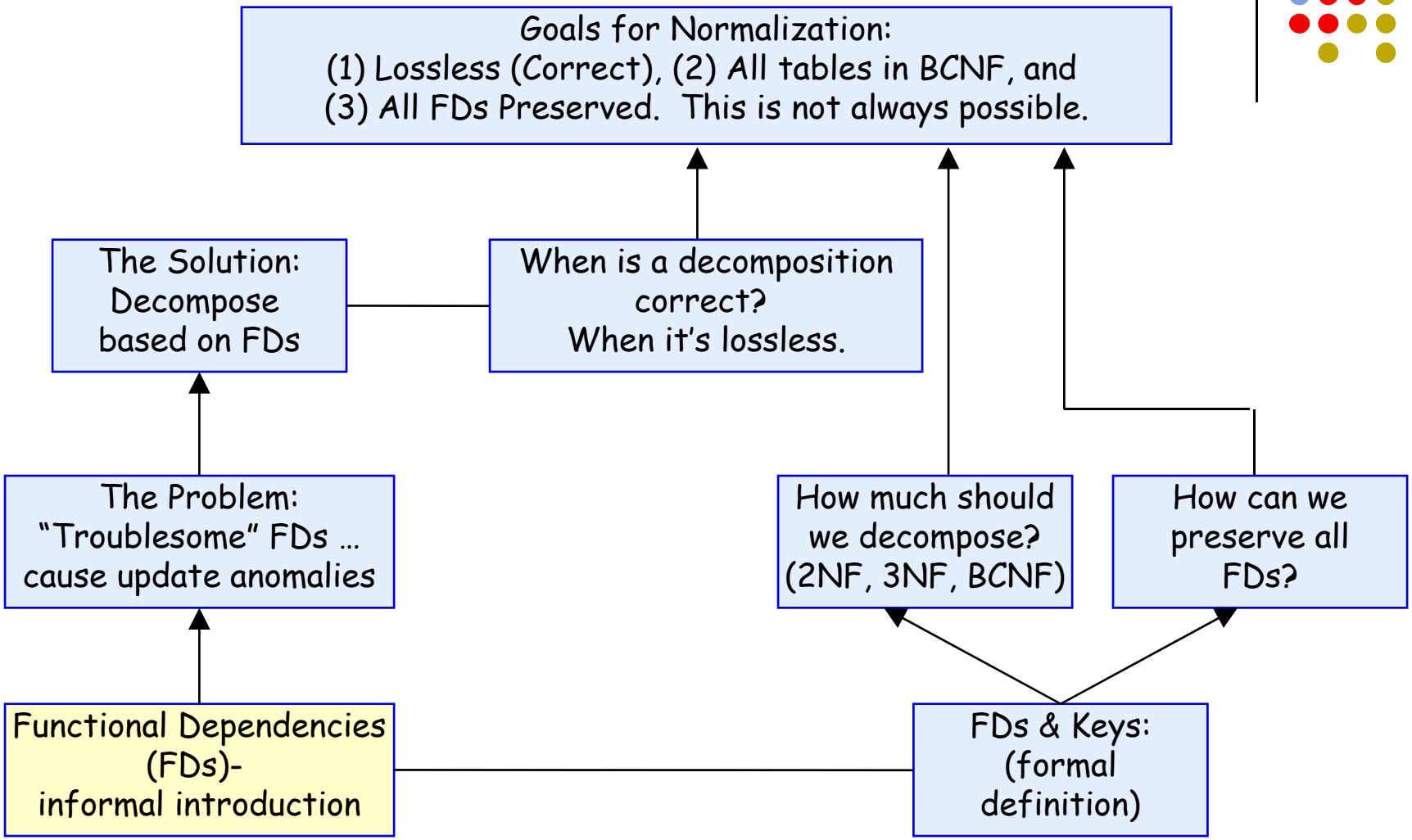
# Normalization Strand Map

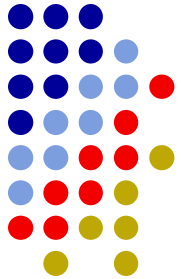





practical aspects

formal aspects

# What is the key for this table?
# What is this table about?



Y(subject, course-num, book-id, faculty-id)

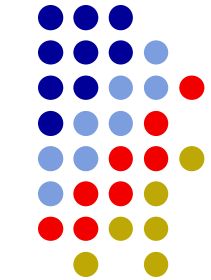
Identifies one course

Identifies one book

Identifies one faculty member

# What is the key for this table?

## Y(subject, course-num, book-id, faculty-id)



Option 1:

Y(subject, course-num, book-id, faculty-id)

Table is about *courses*!

Option 2:

Y(subject, course-num, book-id, faculty-id)

Table tells us *who teaches* which *course*!

Option 3:

Y(subject, course-num, book-id, faculty-id)

Table is about books!

# Notice … only one value for non-key attributes (for each key value)

| Employee | SSN | Name | Salary | Job-code |
|---|---|---|---|---|
| | 111111111 | John Smith | 40,000 | 15 |
| | 123456789 | Mary Smith | 50,000 | 22 |
| | ~~123456789~~ | ~~Marie Jones~~ | ~~50,000~~ | ~~24~~ |

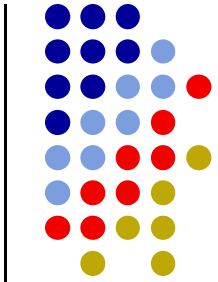
1. NOT allowed because SSN is key!

2. Only one name (and one salary and one Job-code) for each row.

For one particular SSN value, 123-45-6789, there is only ONE name because

1. there is only one tuple and
2. we assume that attributes values are atomic.

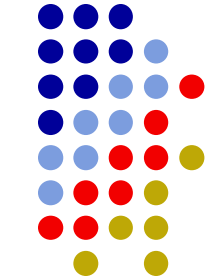
# Definition of a Key for a Relation



A key is a minimal set of attributes in a relation whose values are guaranteed to uniquely identify rows in the relation.

- Two distinct rows have distinct key values (For one key value, there is at most one value for each non-key attribute.)
- (minimal) No subset of the fields that comprise a key is a key

# If you want each code to have one value, use a table with code as key.



For each subject code, there is only one subject. (These codes are excerpted from the PSU Schedule of Classes.)

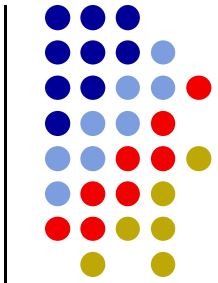
For example, the code “G” is used for Geology (and not for Geography for example).

You can always use a table, with the code and description as attributes, with the code as a key, to enforce this situation.

Subject-Lookup

| Code | Subject |
|---|---|
| CS | Computer Science |
| G | Geology |
| GEOG | Geography |
| HST | History |
| MTH | Mathematics |
| STAT | Statistics |

# Functional Dependencies Intuitive description



An FD, A → B, where A is a set of attributes and B is a set of attributes

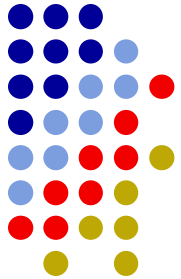
is a statement that if two tuples agree on attributes A they must agree on attributes B

If the values of attributes A are known, then the value of attributes B, are known/determined.

For one A value there is ONLY one B value.

FDs are a generalization of keys.

# Functional Dependencies: Consider the application domain



Likely functional dependencies:

*ssn* → *employee-name*

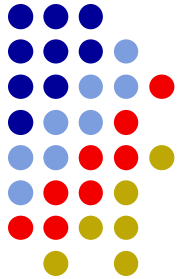
*course-number* → *course-title*

Unlikely functional dependencies

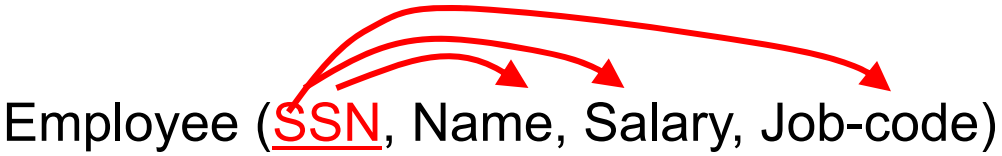
*course-number* $\not\rightarrow$ *book*

*course-number* $\not\rightarrow$ *car-color*

# Every key implies a set of FDs



Each key implies a set of functional dependencies (FDs) from the key to the non-key attributes.




FDs implied by the key:

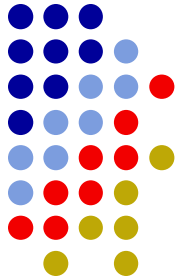
SSN → Name

SSN → Salary

SSN → Job-code

All of these FDs will be enforced when the DBMS enforces the key.

# Keys and FDs (notation)



Given a table R, with a and b (together) as a key for the table, the following FDs are implied by the key.

R (<u>a, b</u>, c, d, e) then

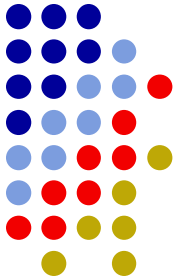
$ab \rightarrow c$

$ab \rightarrow d$

$ab \rightarrow e$

Note we also write this as: $ab \rightarrow cde$

# Functional Dependencies can Suggest Keys



If we know these FDs:

*SSN → name*

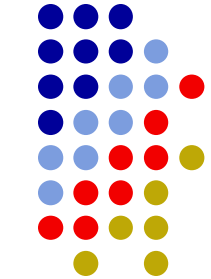
*SSN → hire-date*

*SSN → phone*

then SSN should/must be the key for a table with these attributes:

Employee (<u>SSN</u>, name, hire-date, phone)

# Will FDs (that are not implied by the key) be enforced?



Consider this table:

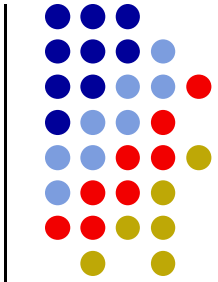
Emp(ssn, name, phone, dept, dept-name)

There is an FD from *dept* → *dept-name*

But *ssn* is the key for this table. dept is NOT the key.

Is it possible for there to be two names for one dept in this database? In other words, will this FD be enforced?

# Will this FD be enforced? Let's try it.



Consider this table:

Emp(ssn, name, phone, dept, dept-name)

| Employee | ssn | Name | Phone | Dept | Dept-name |
|---|---|---|---|---|---|
| | 111111111 | John | 555-1234 | 12 | Sales |
| | 222222222 | Mary | 555-7890 | 12 | Marketing |
| | … | | | | |

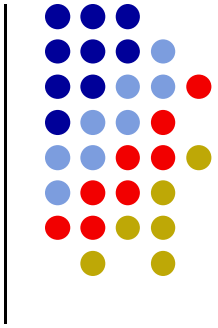
Can we put these two rows in this table?

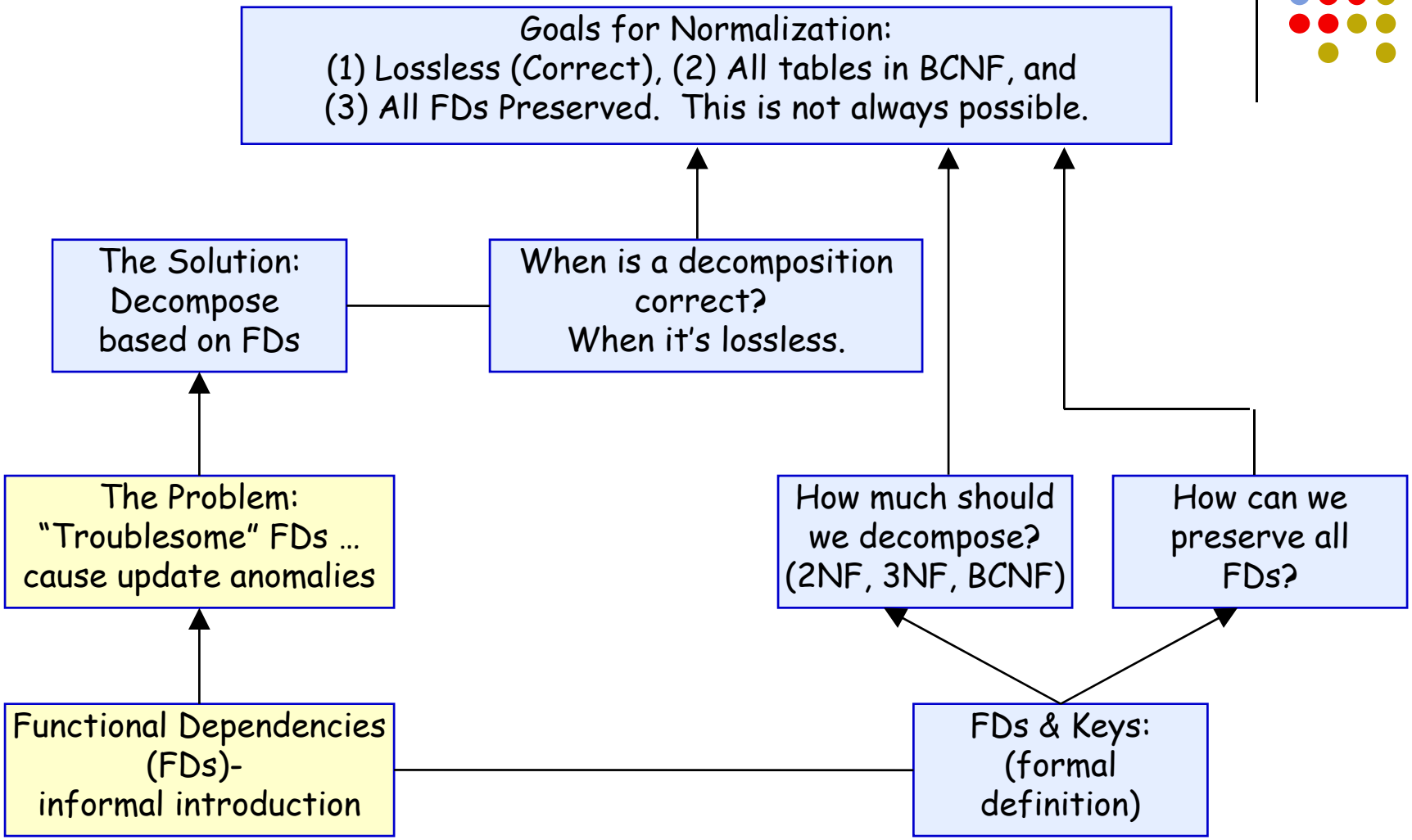
Yes, it doesn't violate the key constraint.

But, the FD from dept to dept-name is violated! We shouldn't haven't two different names for dept 12!

We might say, informally, that *dept* "ought to be a key".

# Normalization Strand Map





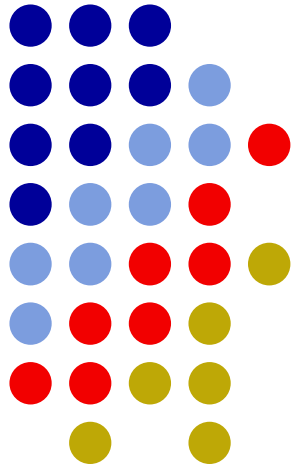

practical aspects

formal aspects

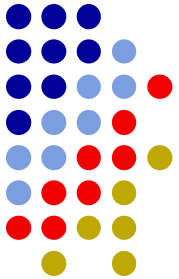
# The Problem: "Troublesome" FDs (Motivation for normalization)

"Troublesome" FDs cause redundancy and update anomalies.

"Troublesome" FDs = FDs that are NOT implied by the keys for the relation.



# Redundancy (storing info. multiple times)

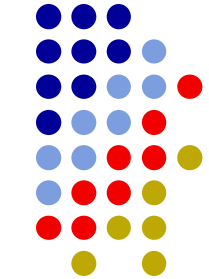


- Disadvantage: Any time information is stored more than once, it has the possibility of being inconsistent.
  - Phone numbers in your handheld
  - Phone numbers in your cell phone
  - Phone numbers in your address book

If someone changes their phone number, do you remember to change it in every place?

- Advantage: Redundant copies improve retrieval/queries!

# Sometimes Redundancy is Caused by one or more FDs



Consider this table:

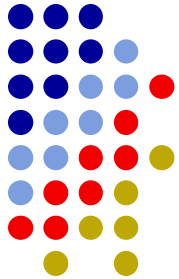
EMP(name, SSN, birthdate, address, dnum, dname, dmgr)

Then *dname* and *dmgr* are stored redundantly – whenever there are multiple employees in a department.

This redundancy is caused by what I informally call “troublesome” FDs. The FDs shown in blue are “troublesome”.

Note: foreign keys always carry redundant values. We can’t get rid of this kind of redundancy.

# Redundancy Caused by Troublesome FD – Sample Data



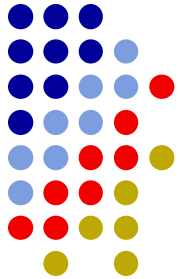
EMP(name, SSN, birthdate, address, dnum, dname, dmgr)

| | | | | | | |
|---|---|---|---|---|---|---|
| John | 111 | June 3 | 123 St. | D1 | sales | 222 |
| Sue | 222 | May 15 | 455 St. | D1 | sales | 222 |
| Max | 333 | Mar. 5 | 678 St. | D2 | research | 333 |
| Wei | 444 | May 2 | 999 St. | D2 | research | 333 |
| Tom | 555 | June 22 | 888 St. | D2 | research | 333 |

We have the department name and manager twice for D1 and three times for D2!

# Update anomalies for

EMP(name, SSN, birthdate, address, dnum, dname, dmgr)



Insertion anomalies:

if you insert an employee with a department
then you need to know the descriptive information for that department.

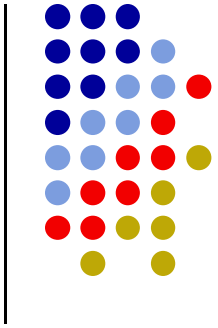
if you want to insert a department, you can't ... until there is at least one employee.

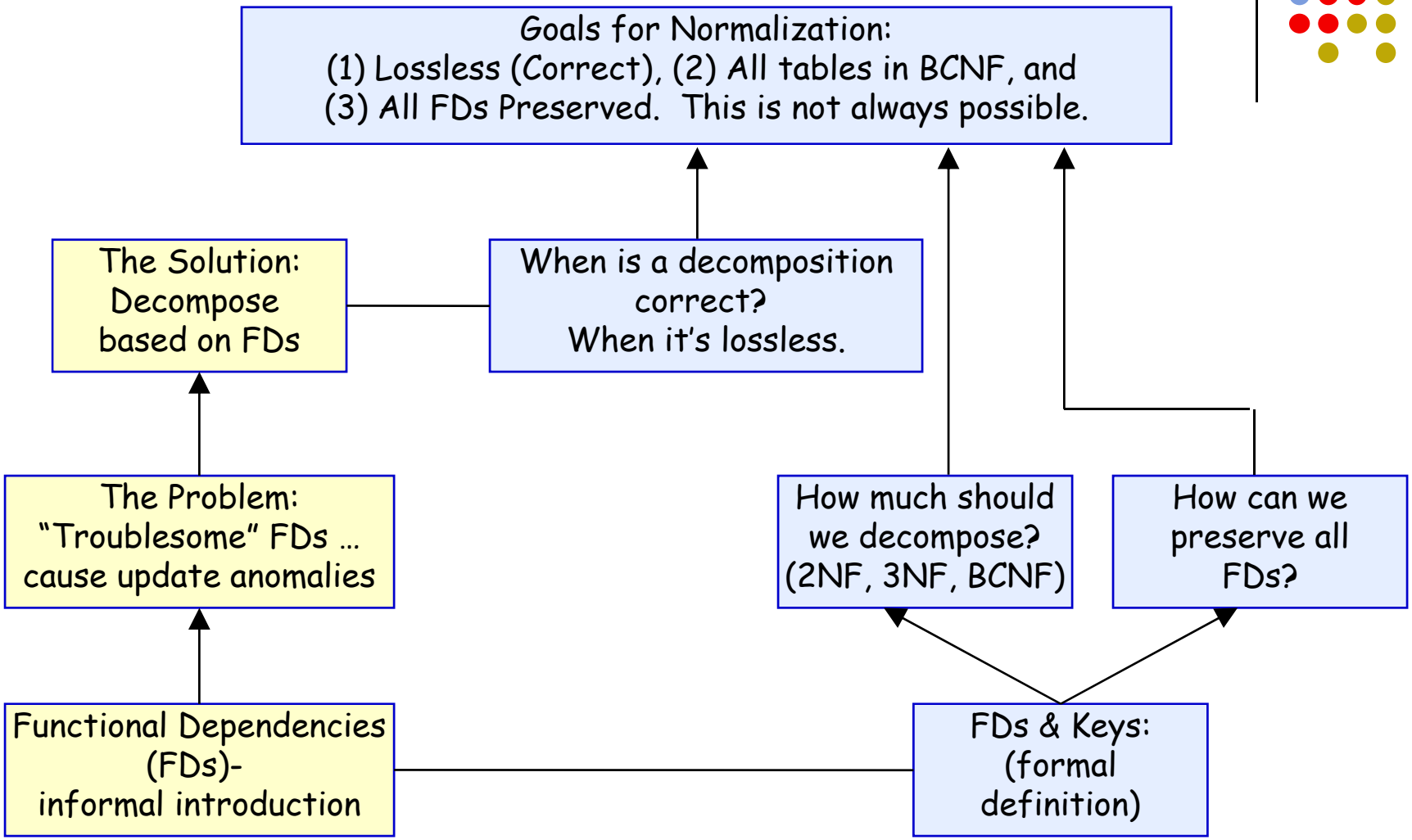
Deletion anomalies: if you delete an employee, is that dept. gone? Was this the last employee in that dept.?

Modification anomalies: If you want to change *dname*, for example, you need to change it everywhere! And you have to find them all first.

Troublesome FDs cause (redundancy and) update anomalies.

# Normalization Strand Map






practical aspects

formal aspects

# The Solution: Decompose relations with "troublesome" FDs into two (or more) relations

Normalization by decomposition based on FDs,

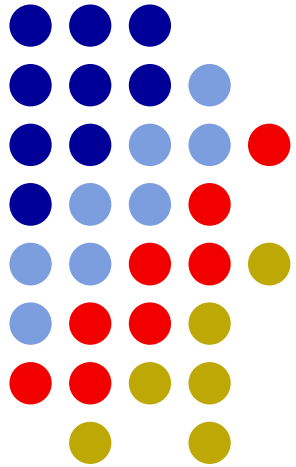
reduces redundancy,

reduces the pressure to use null values,

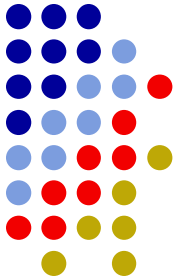
and, most importantly,

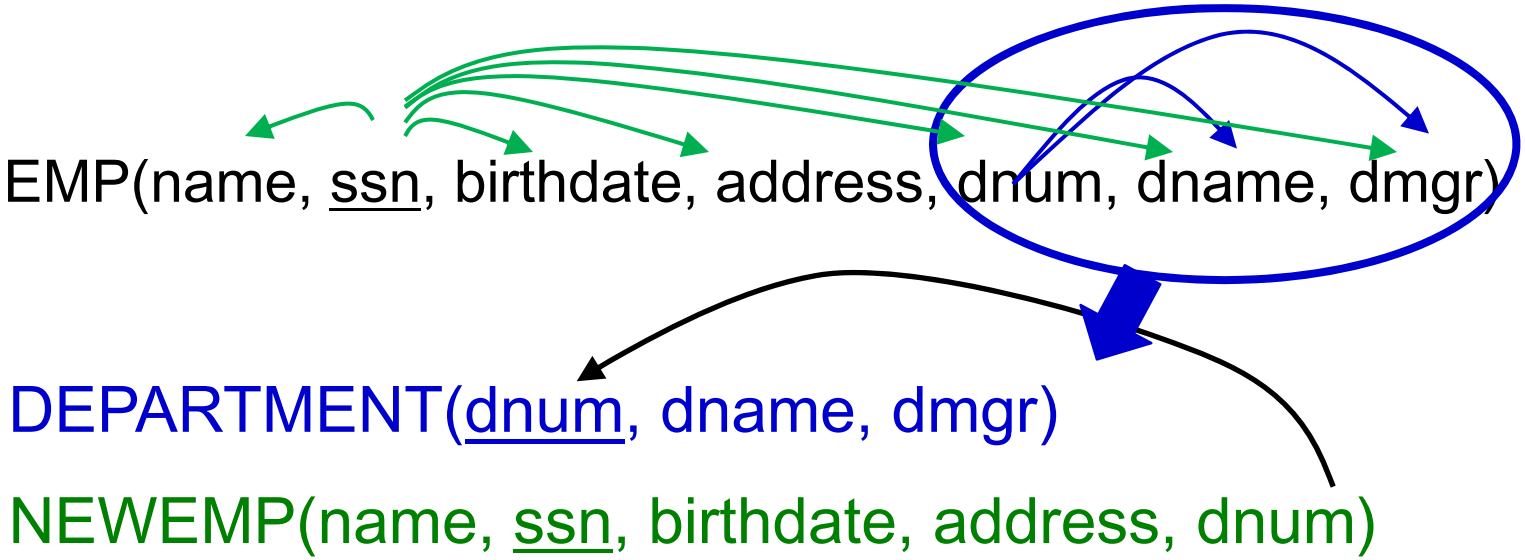
eliminates update anomalies.

Also, all FDs will be enforced (just by keys).



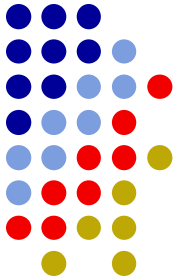
# Example: Decompose relations with troublesome FDs into two






1. Lift the “troublesome” FDs into their own table with dnum as the key. Now they will be enforced.
2. Leave the LHS of the “troublesome” FDs behind. Define a foreign key where Employee.dnum REFERENCES Department.dnum

# Advantages of Normalization based on Decomposition



When this table:

Emp(name, ssn, birthdate, address, dnum, dname, dmgr)

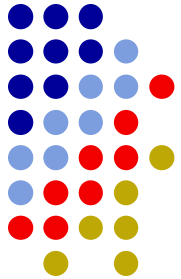
is replaced by these two tables:

Department(dnum, dname, dmgr)

NewEmp (name, ssn, birthdate, address, dnum)

Are there any update anomalies in the new tables?

# Let's Check



Department(dnum, dname, dmgr)

NewEmp (name, ssn, birthdate, address, dnum)

Insertion anomalies:

if you insert an employee with a department then you need to know the descriptive information for that department. NO – ONLY THE NUMBER

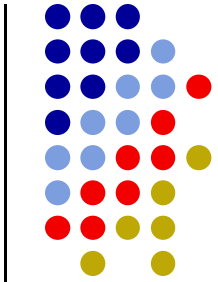
if you want to insert a department, you can't ... until there is at least one employee. NO PROBLEM

Deletion anomalies: if you delete an employee, is that dept. gone? Was this the last employee in that dept.? NO PROBLEM

Modification anomalies: If you want to change *dname*, for example, you need to change it everywhere! And you have to find them all first. dname is only stored once!

Is there any redundancy? Yes – in the foreign key.

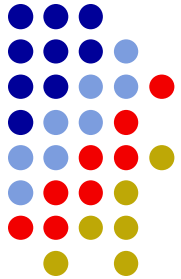
# Basic Idea: Normalize (decompose) based on FDs

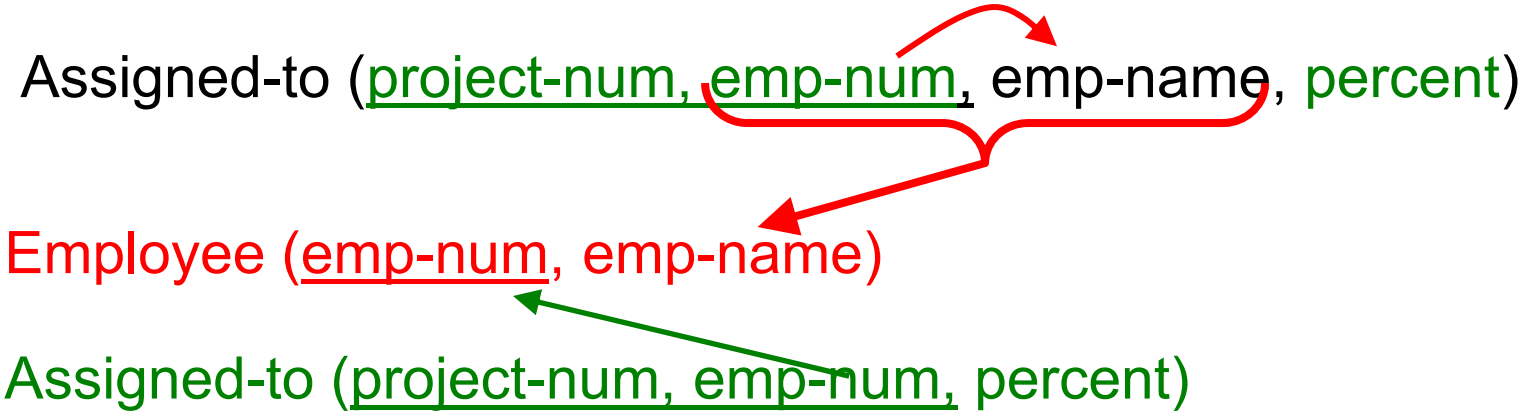


- Identify all the (non-trivial) FDs in an application.
  - Identify FDs that are implied by the keys.
  - Identify FDs that are NOT implied by the keys – the "troublesome" ones.

- Decompose a table with a "troublesome" FD into two or more tables by "lifting" the troublesome FDs into a table of their own. Note: when there are two or more "troublesome" FDs with the same LHS, then they can be lifted, together, into a single table.

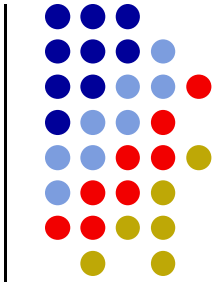
# Another Decomposition Example





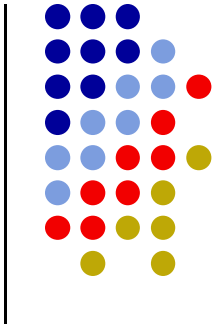

1. Lift the troublesome FD(s) into a table of their own.
   Key for new table is left hand side of the troublesome FD.
2. Leave the left side of the FD behind in the original table.
3. Eliminate *emp-name* from the *Assigned-to* table.

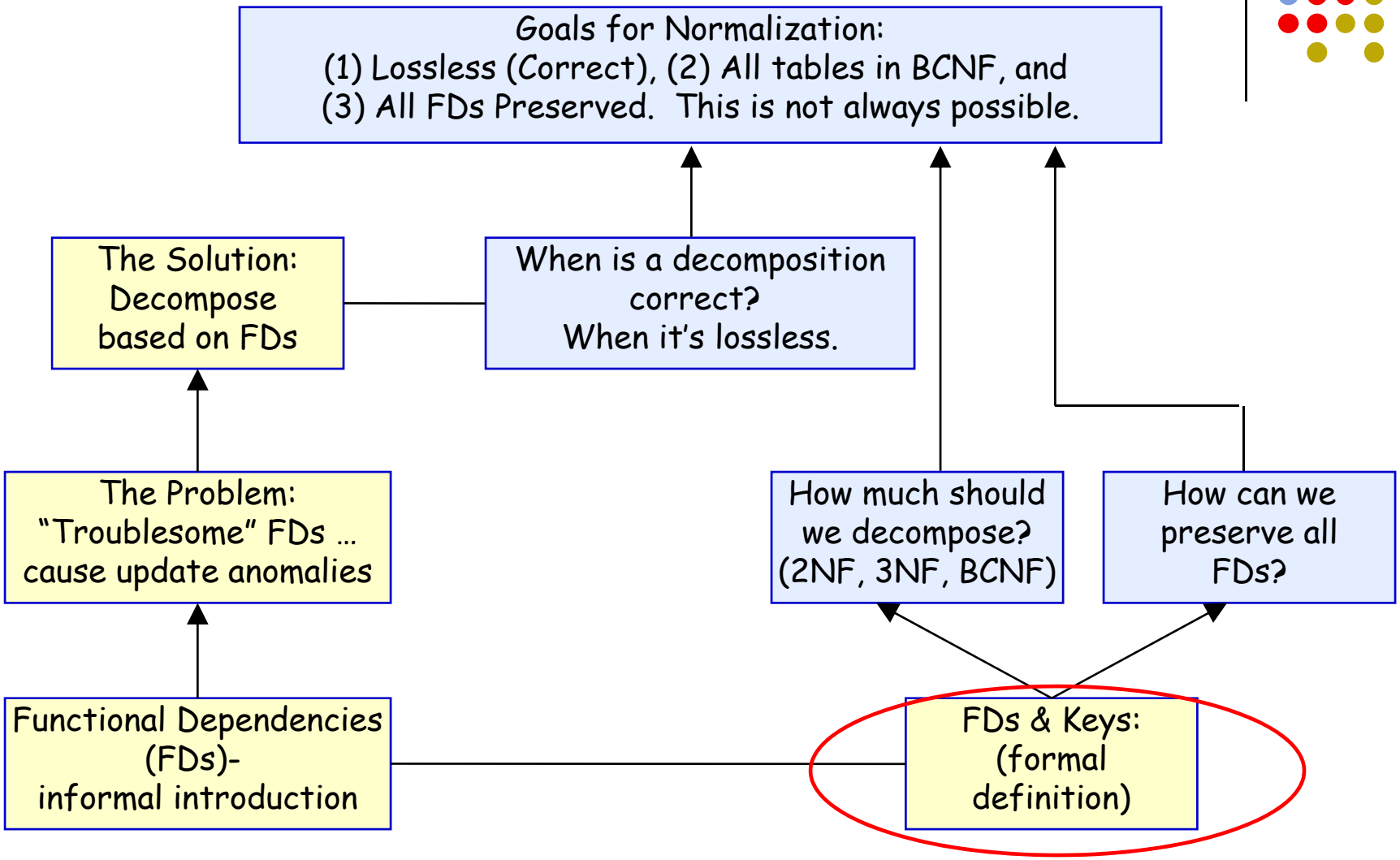
# Questions about normalization



- How do we know which FDs we have?
  Talk to domain experts; identify FDs; use them as the starting point for normalization.
- How do we know if the decomposition is correct?
- How do we know how much to normalize?
  How far should we go?
- How do we know if all of the FDs of interest are being enforced – by using keys for a table?
  We need the formal definition of FDs to be able to answer these last three questions.

# Normalization Strand Map



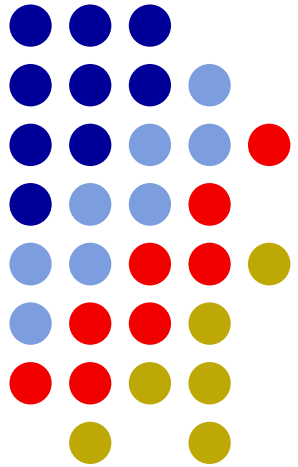



practical aspects

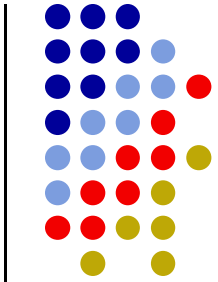
formal aspects

# FDs and Keys: Formal Definition

A functional dependency is formally defined as a functional relationship between two sets of attributes. This leads to the definition of trivial FDs and superkeys.



# Definition of a function



Remember the definition of a function:

| x | f(x) |
|---|---|
| 1 | 2 |
| 1 | 3 |
| 2 | 5 |
| 3 | 5 |

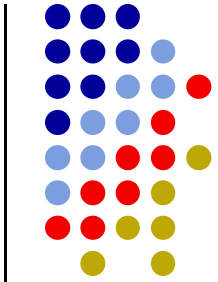
| x | g(x) |
|---|---|
| 1 | 2 |
| 2 | 2 |
| 3 | 5 |

| x | h(x) |
|---|---|
| 1 | 10 |
| 2 | 20 |
| 3 | 30 |

Which of these are functions?

An FD is a functional relationship
(that **must** occur in a relation) among attribute values

# Answer (concerning functions)



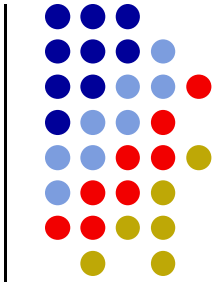
| x | f(x) |
|---|---|
| 1 | 2 |
| 1 | 3 |
| 2 | 5 |
| 3 | 5 |

| x | g(x) |
|---|---|
| 1 | 2 |
| 2 | 2 |
| 3 | 5 |

| x | h(x) |
|---|---|
| 1 | 10 |
| 2 | 20 |
| 3 | 30 |

f is NOT a function because for an input of “1” there are two answers, “2” and “3”.
g and h are functions.
h is a one-to-one (injective) function.

# Example of an FD – a function



Employee (ssn, name, phone, salary)

Since *ssn* → *name* is an FD

we know that there is only one name for an ssn.

Thus we know that *ssn and name* are in a functional relationship!

Note: we still need to store this relation on disk; there is no way to compute name from ssn.

# Another Example

Employee (ssn, name, phone, dept, dept-mgr)

*dept → dept-mgr*

we know that there is only one dept-mgr for a dept.

We know that *dept → dept-mgr* is a function!

# Trivial FD

We have a trivial FD whenever the attributes on the right side of an FD are a subset of the attributes on the left side of the FD:

$A \rightarrow A$

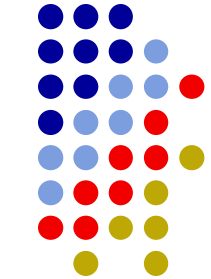
$AB \rightarrow A$

$ABCD \rightarrow BCD$

A trivial FD represents an function: f(X) = X

It is definitely a function … and definitely an FD. But it's not "troublesome" and won't help us decompose a table. There are LOTS of trivial FDs.

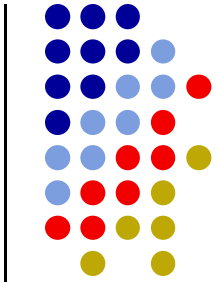
# Definition of a Key for a Relation (repeated from earlier)



A key is a minimal set of attributes in a relation whose values are guaranteed to uniquely identify rows in the relation.

- Two distinct rows have distinct key values
- (minimal) No subset of the fields that comprise a key is a key

# Definition of a Superkey for a relation



A *superkey* is a set of fields from a relation that contains a key.

Every key is (automatically) a superkey.

A superkey is NOT necessarily a key.

Example:

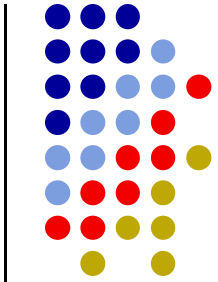
Emp (SSN, name, phone, dept)

SSN is a key for this relation.

(dept, SSN) is a superkey for this relation.

Challenge question: For an arbitrary relation, can you name a superkey?

# Keys and FDs are Constraints

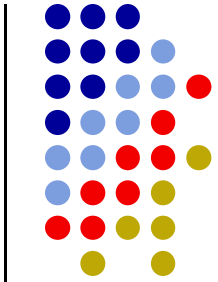


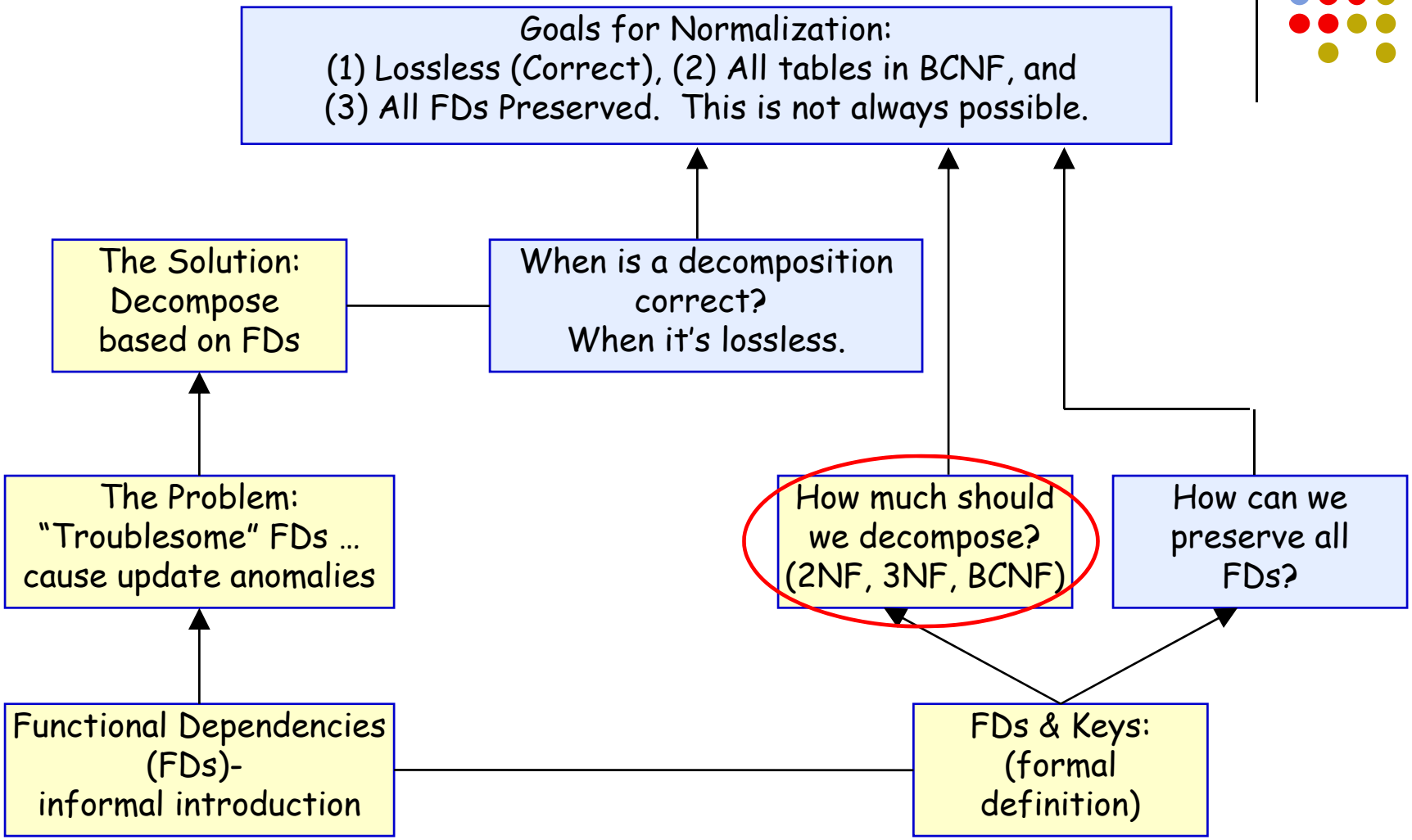
- We need to know if keys and FDs (always) hold in the application.

- We need to consult a domain expert to find out what the keys and FDs are. The keys and FDs serve as input to the database design process.

  That is, we (the DB designers) don't get to decide whether or not an FD or a key holds.

# Normalization Strand Map






practical aspects

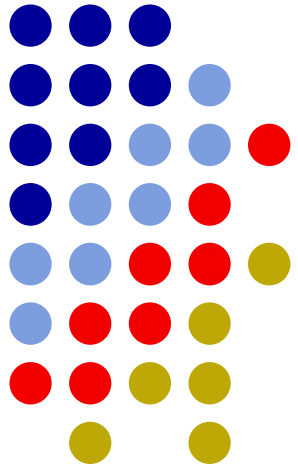
formal aspects

# 2NF, 3NF, BCNF: Normal forms based on FDs

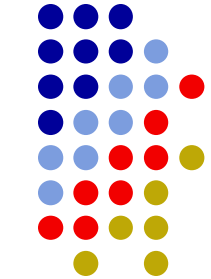
Given a set of FDs and one or more tables, three increasingly stronger normal forms, namely 2NF, 3NF, and BCNF, have been defined.

BCNF implies 3NF.

3NF implies 2NF.



# Informal Definitions
# Normal Forms Based on FDs



1NF - all attribute values (domain values) are atomic
(part of the definition of the relational model)

---

2NF - all non-key attributes must depend on the **whole** key (no partial dependencies)

r (A B C D E) B → C violates 2NF

3NF – table is in 2NF and all non-key attributes must depend on **only** a key (no transitive dependencies)

r (A B C D E)

BCNF - every determinant (LHS of an FD) is a key for the table (All FDs are implied by the keys)

r (A B C D E)

# Examples of Violations

2NF - all non-key attributes must depend on a whole key

Assigned-to (A-project, A-emp, emp-name, percent)

3NF - all non-key attributes must depend on only a key

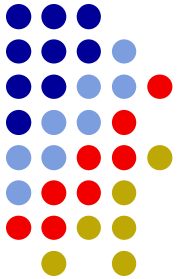
Employee (SSN, name, address, project, p-title)

BCNF - every determinant (LHS of an FD) is a key for this table

(all FDs are implied by the keys)

Assigned-to (Emp-ID, A-Project, SSN, percent)

# Fix violations of 2NF by lifting "troublesome" FDs



(Example repeated from earlier slide)

Assigned-to (A-project, A-emp, emp-name, percent)

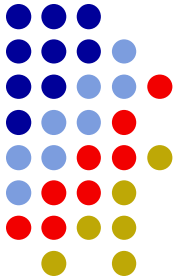
Employee (A-emp, emp-name)

1. Lift the troublesome FD(s) into a table of their own.
   Key for new table is left hand side of the troublesome FD.

2. Leave the left side of the FD behind in the original table.
   Assigned-to (A-project, A-emp, percent)

3. Eliminate *emp-name* from the *Assigned-to* table.

# Fix violations of 3NF by "lifting" troublesome FDs



(Example repeated from earlier slide)

Emp(name, ssn, birthdate, address, dnum, dname, dmgr)

Department(dnum, dname, dmgr)

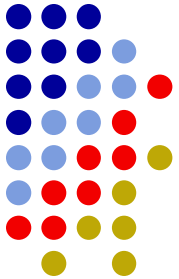
1. Lift the troublesome FD(s) into a table of their own.
   Key for new table is left hand side of the troublesome FD.

2. Leave the left side of the FD behind in the original table.

NewEmp (name, ssn, birthdate, address, dnum)

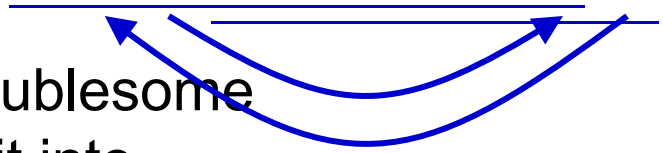
3. Eliminate *dname* and *dmgr* from the *NewEmp* table.

# Fix violations of BCNF by lifting "troublesome" FDs



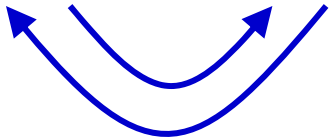
BCNF - every determinant is a key (all FDs implied by the keys)

Assigned-to (A-emp, A-project, A-ssn, percent)



Take the troublesome
FD and put it into
a table of it's own.

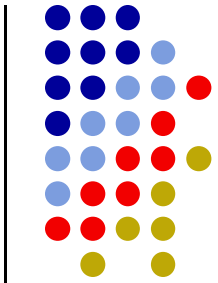
Assigned-to (A-emp, A-ssn)



Then leave the left side of the troublesome FD in the original table:

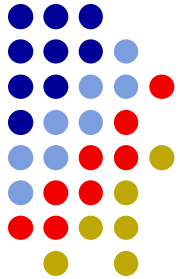
Assigned-to (A-emp, A-project, percent)

# Formal definition of 3NF
## (in the textbook)



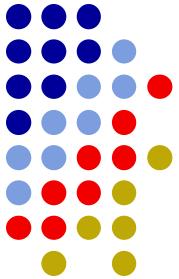
- For a table R, every FD X → A that occurs among attributes of R then either:
  - A is an element of X (X → A is trivial)
  - A is part of a key (ignore the "key" attributes)
  - X is a superkey of R
    Consider the following 2 cases:
    - X is a key for R (good)
    - X is a superkey for R (and not a key). The X → A is derivable from a key using augmentation. (Stay tuned.)

# Formal definition of BCNF
## (in the textbook)



- For a table R, every FD X → A that occurs among attributes of R then either:
  - A is an element of X (X → A is trivial)
  - ~~A is part of a key (don't worry about "key" attributes)~~
  - X is a superkey of R
    consider the following 2 cases:
    - X is a key for R (good)
    - X is a superkey for R (and not a key). The X → A is derivable from a key using augmentation. (Stay tuned.)

# One Goal - BCNF

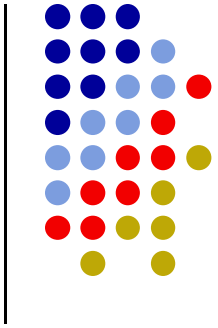


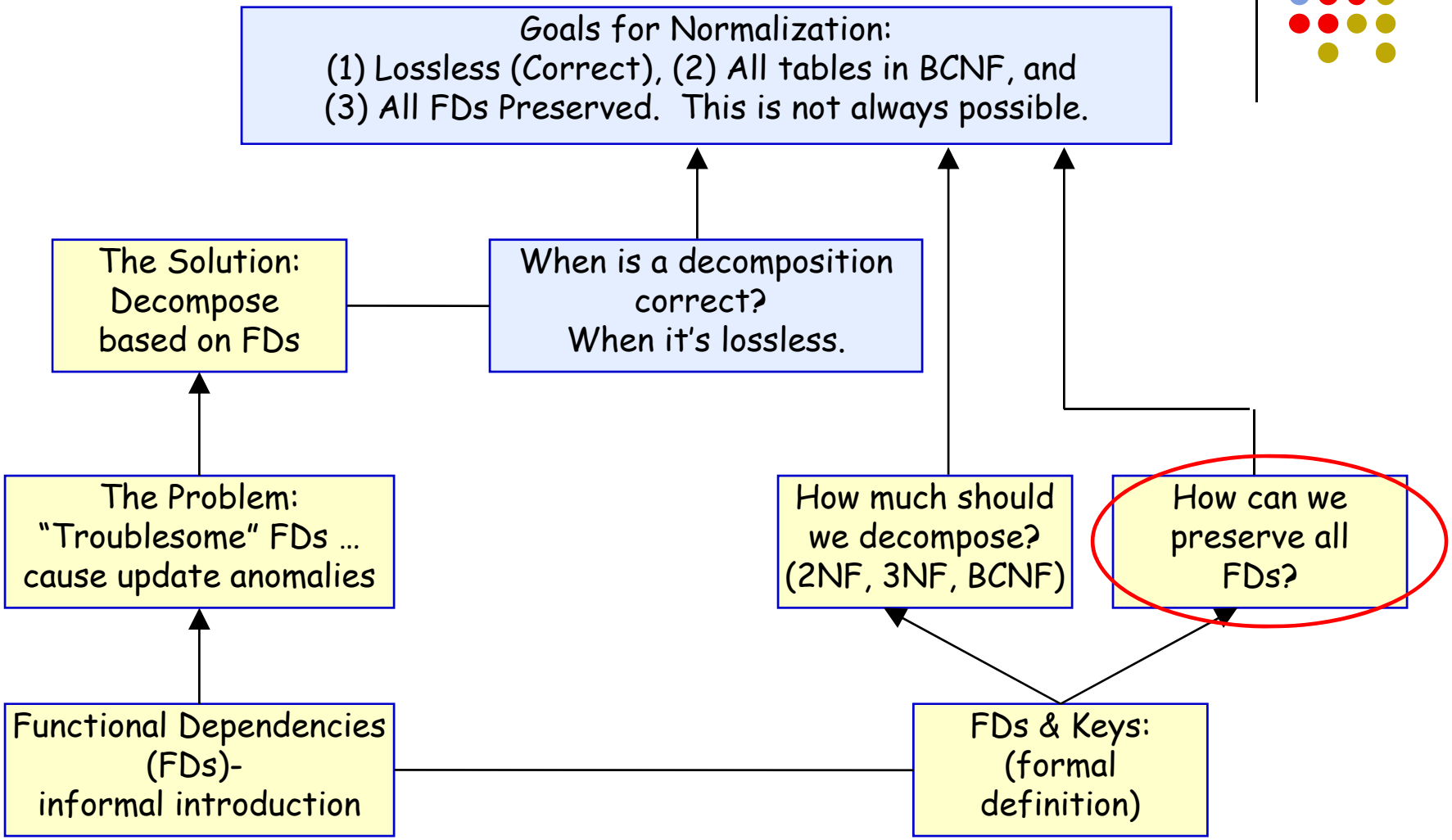
Whenever we normalize tables, we would like to decompose until all tables are in BCNF.

Then, there are no remaining redundancies (and update anomalies) caused by FDs.

That is, we want all FDs implied by the keys.

# Normalization Strand Map



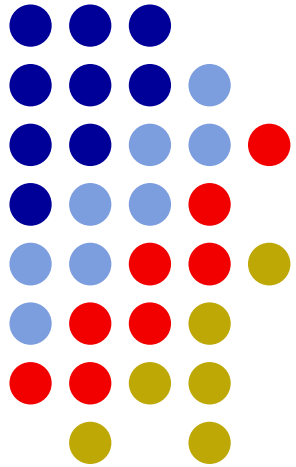



practical aspects

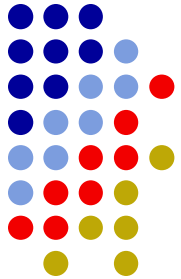
formal aspects

# Dependency Preservation: Using a sound & complete set of inference rules

Dependency preservation requires the use of a sound and complete set of rules of inference to compute $F^+$, the closure of a set F of FDs. Given the original set of FDs, F. Then G is the set of FDs that are implied by the keys in the resulting decomposition. Dependency preservation is when $F^+ = G^+$.



# When we decompose relations, we must project the FDs



When we decompose a relation into smaller relations,

Employee (SSN, name, phone, dept, dept-name)

Into:

Employee (SSN, name, phone, dept)

Department (dept, dname)

We must check each FD in the original set (plus more)

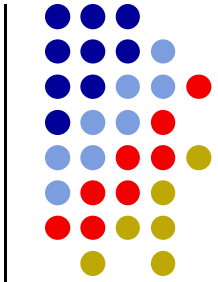
SSN→name SSN→phone SSN→dept

SSN→dept-name dept→dept-name

We project the set of FDs by retaining only those where all of the attributes in the FD are in one table.

SSN→name, SSN→dept, and dept→dept-name are fine but SSN→dept appears to be lost, for example.

# Losing an FD



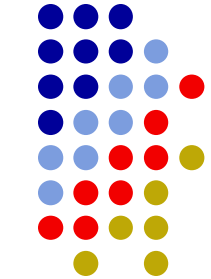
If I have an FD dept → mgr

and if dept ends up in one table and
mgr ends up in another table

Then this FD will NOT be enforced.

That's why we project a set of FDs onto the tables that we end up with, after decomposition. We need to see whether they are still there (and whether they are implied by the key … that is, that the table is in BCNF).

# Example: do we lose an FD?



Employee (SSN, name, phone, dept, dept-name)

Original FDs F:

SSN→name SSN→phone SSN→dept

SSN→dept-name dept→dept-name

Employee (SSN, name, phone, dept)

Department (dept, dname)

Resulting FDs G:

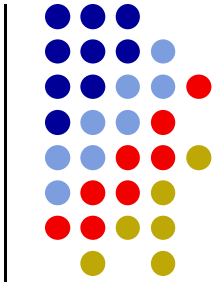
SSN→name SSN→phone SSN→dept

dept→dept-name

What about SSN→dept-name? Is it lost? Can there be two department names for one SSN?

NO! It's not lost. One SSN has only one dept. And one dept has only one dept-name. So SSN has only one dept-name.

# Inference rules to derive all FDs from a given set of FDs.



For sets of attribute X and Y

Reflexivity

If Y is a subset of X, then X $\rightarrow$ Y

examples: name $\rightarrow$ name, gender $\rightarrow$ gender
name, shoe-size $\rightarrow$ name

Augmentation

for all FDs, A $\rightarrow$ B then AC $\rightarrow$ BC, for all C

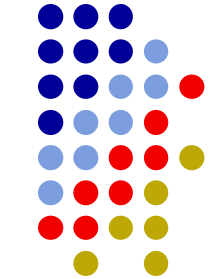
example: augmentation creates superkeys from keys.

Transitivity

for all FDs, if A $\rightarrow$ B and B $\rightarrow$ C

then A $\rightarrow$ C

# Use the rules to compute closure



Let F be a set of FDs.

$F^+$ is the set of all FDs implied (or derivable) from F

using a sound & complete set of inference rules

Reflexivity: If Y is a set of attrs, X subset of Y, then $X \rightarrow Y$

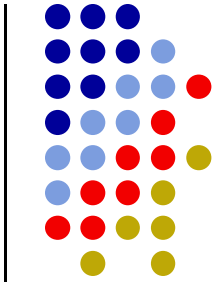
Augmentation: If $X \rightarrow Y$, and Z is a set of attrs, then $ZA \rightarrow ZB$

Transitivity: If $X \rightarrow Y$, $Y \rightarrow Z$, then $X \rightarrow Z$

Compute $F^+$ by applying rules until no new FDs arise.

$F^+$ is called the closure of F.

# Definition of Dependency Preserving



Suppose F is the original set of FDs.

Compute F+.

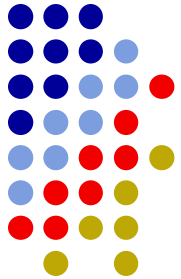
G is set of FDs from $F^+$ that are present in individual relations in G. (We project F+ to relations in the resulting, decomposed schema.)

Compute $G^+$.
If $F^+ = G^+$

then the decomposition is dependency preserving

For a complex design, you may want to implement one of the known algorithms for computing $F^+$ and $G^+$.

# When we decompose relations, we must project the FDs (repeated)



When we decompose a relation into smaller relations,

Employee (SSN, name, phone, dept, dept-name)

Into:

Employee (SSN, name, phone, dept)

Department (dept, dname)

Check each FD in $F^+$

We must check each FD in the original set (plus more)

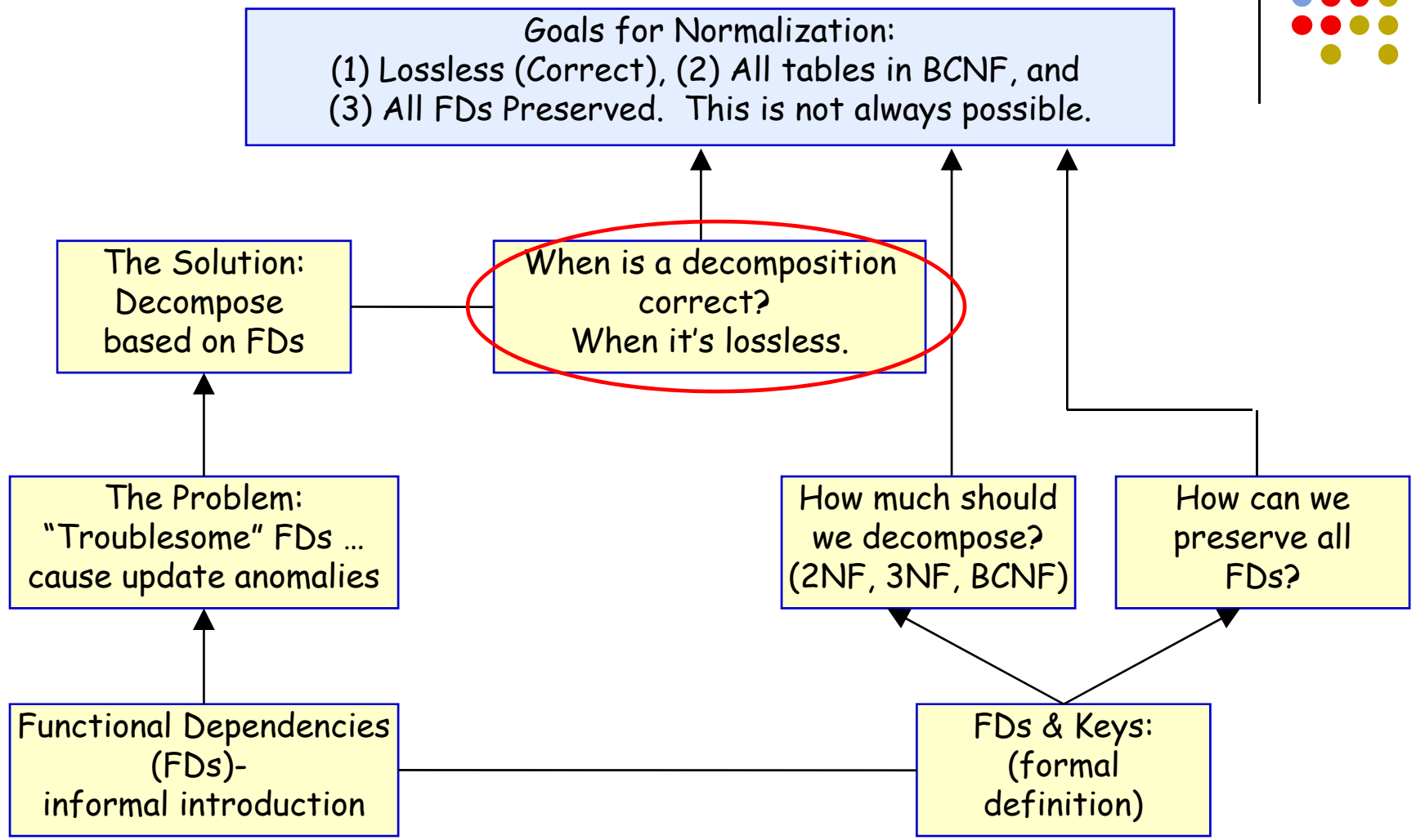
SSN→name SSN→phone SSN→dept

SSN→dept-name dept→dept-name

We project the set of FDs by retaining only those where all of the attributes in the FD are in one table.

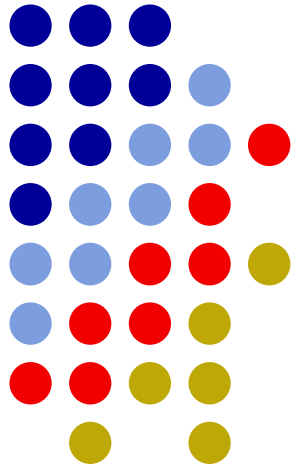
SSN→name, SSN→dept, and dept→dept-name are fine but SSN→dept appears to be lost, for example.

# Normalization Strand Map

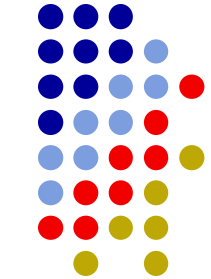


# Decomposition is correct when it is lossless

The decomposition algorithm (based on lifting "troublesome" FDs into a separate table) guarantees that the decomposition of the original table is lossless.



# Decompose: Project Operator
# Recompose: Join Operator



When

Emp(name, SSN, birthdate, address, dnum, dname, dmgr)

is replaced by these two tables:

Department(dnum, dname, dmgr)

NewEmp (name, SSN, birthdate, address, dnum)

We use the project operator to decompose

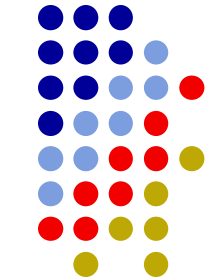
$$\text{Department} = \pi_{\text{dnum,dname,dmgr}}\text{Emp}$$

$$\text{NewEmp} = \pi_{\text{name,SSN,birthdate,address,dnum}}\text{Emp}$$

And we use the join operator to put the pieces together

$$\text{Emp} = \text{Department} \; |X|_{\text{D.SSN=NewEmp.SSN}} \; \text{NewEmp}$$

# What is a lossless (and a lossy) decomposition?



We want to make sure that we haven't thrown away any information from the original schema.

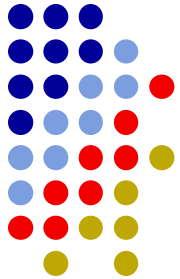
When table R is decomposed into tables R1 and R2 then the decomposition is lossless (correct) if:

(R1 |X| R2) is identical to R

natural join

If it is a lossy decomposition, then R1 |X| R2 gives you TOO MANY tuples.

# Example: a lossy decomposition



original

Employee(SS-number, name, p-num, p-title)

| | | | |
|---|---|---|---|
| 1 | smith | p1 | accounting |
| 2 | jones | p1 | accounting |
| 3 | smith | p2 | billing |

decomposition:

Employee (SS-number, name)

| | |
|---|---|
| 1 | smith |
| 2 | jones |
| 3 | smith |

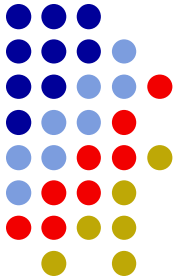
Project (p-num, p-title, name)

| | | |
|---|---|---|
| p1 | account | smith |
| p1 | account | jones |
| p2 | billing | smith |

now with natural join: you get at least one extra tuple!!!

| | | | |
|---|---|---|---|
| 1 | smith | p2 | billing |

# Example: Test for a Lossless Decomposition



Consider a table:

R (a, b, c, d, e) with a troublesome FD d→e.

Decompose it into two tables:

R1(a, b, c, d)

R2(d, e)

As long as

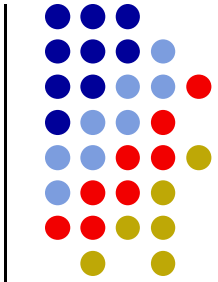
the attributes in common are a key for (at least) one of the relations, $R_1$ or $R_2$

then we know that the decomposition is lossless!

For this example d is the attribute in common.

And d is a key for R2, the second table.

# Is the Decomposition Algorithm Lossless?



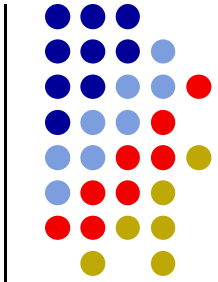
1. Lift the “troublesome” FD(s) (all the FDs with the same LHS) into a table of their own. Key for new table is left hand side of the troublesome FD(s).

2. Leave the left side of the FD behind in the original table.

3. Eliminate the RHS attributes from the original table.

Yes, we are guaranteed that the decomposition is lossless. The attribute in common is definitely a key for the new “lifted” table.

# Example of a Lossy Decomposition (revisited)

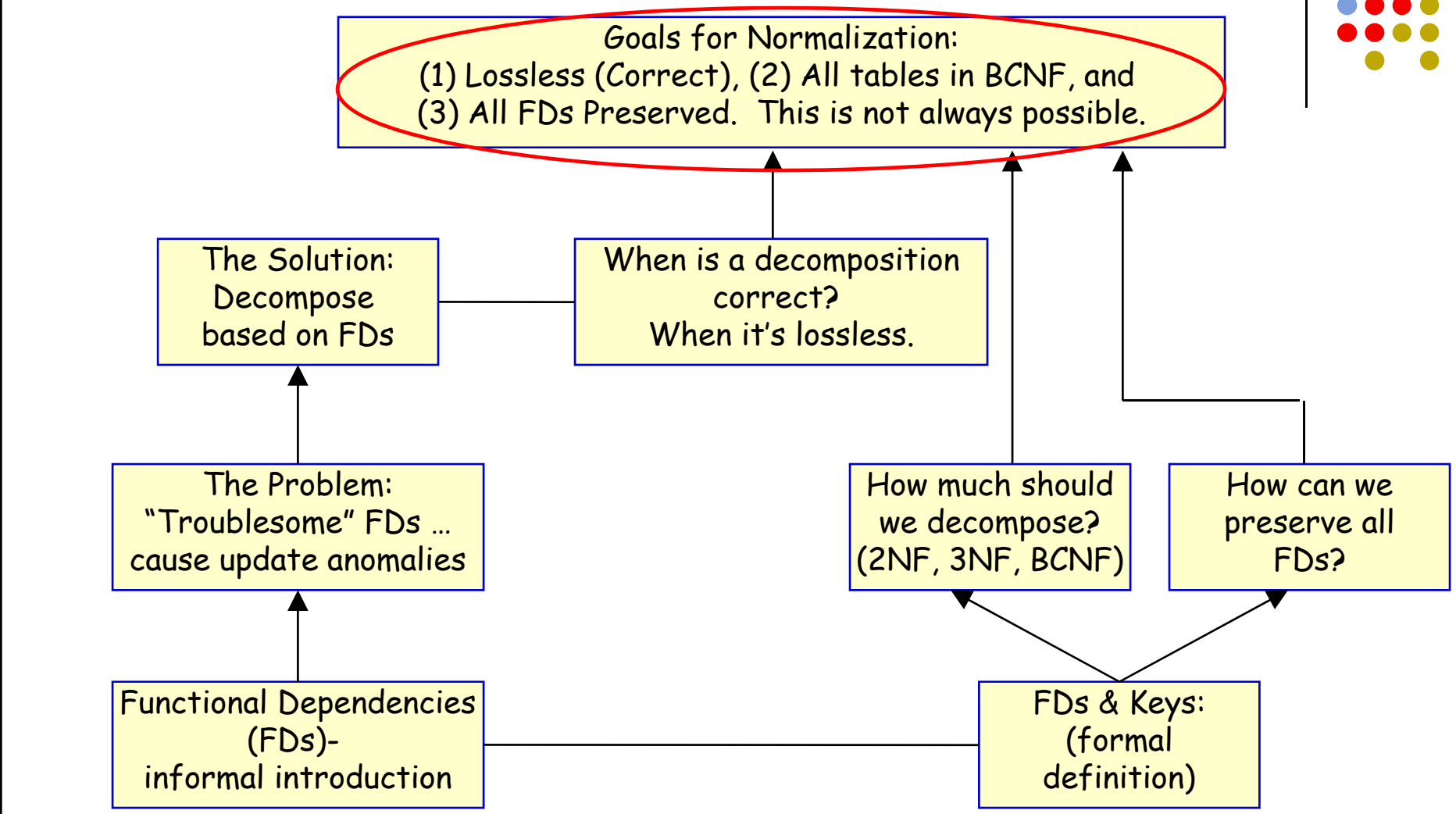


Employee(SS-number, name, project, p-title)

decomposition: Employee (SS-number, name)
Project (p-num, p-title, name)

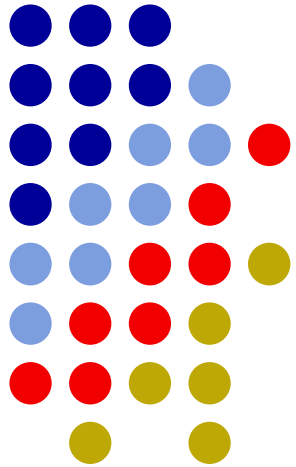
Notice that the common attribute, name, is not a key for either of these tables.

# Normalization Strand Map

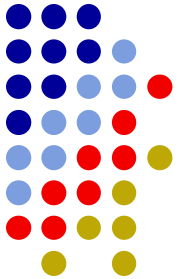


# Three Goals for Normalization: Lossless, BCNF, Dep. Preserving Decomposition

Given a set of FDs and the original set of relations, the goals for normalization (using lossless decomposition based on FDs) are to have all resulting tables in BCNF and all FDs preserved.



# Three Goals for Normalization



lossless decomposition

don't throw any information away

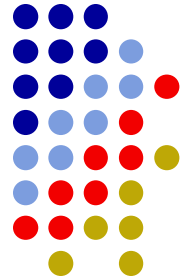
be able to reconstruct the original relation

dependency preservation

all of the original, non-trivial FDs can be derived from FDs implied by the keys of resulting tables

Boyce-Codd normal form (BCNF) - no redundancy beyond foreign keys; all FDs implied by keys

# It is not always possible to have BCNF AND dependency preservation

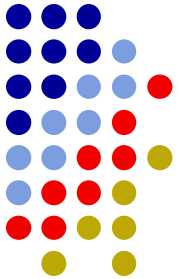


Required! → lossless decomposition

Desirable but not always possible to have both → dependency preservation

Desirable but not always possible to have both → Boyce-Codd normal form (BCNF)

# Counterexample
## (a table that can't be decomposed into BCNF with dependency preservation)



Original table – a table that holds US addresses

addr(number street city state zip)

The original FDs are:

*number street city state -> zip*

*zip -> state*

# Counterexample (cont.)

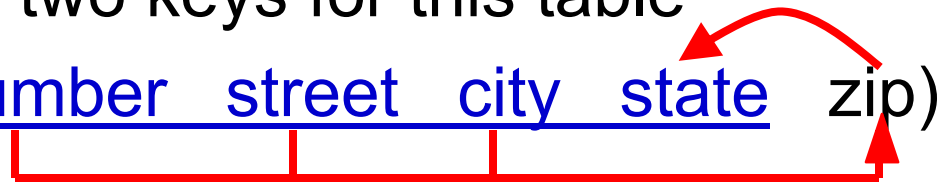
Based on the FDs:

*number street city state -> zip*

*zip -> state*

There are two keys for this table

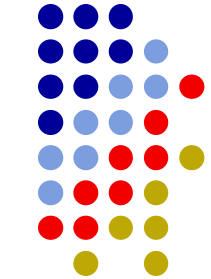
addr(number street city state zip)



Since all attributes are key attributes, this table is automatically in 3NF and 2NF.

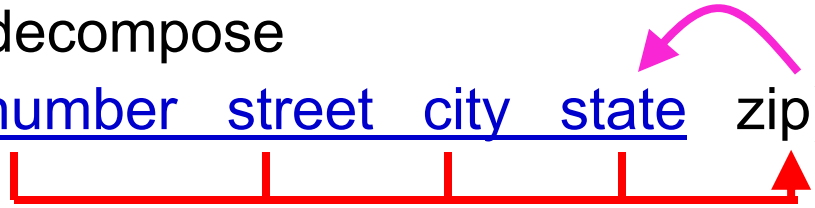
But *zip* → *state* violates BCNF

# Counterexample (cont.)



Let's decompose

addr(number street city state zip)



using this "troublesome" FD:

*zip -> state*

Addr2 (number, street, city, zip)

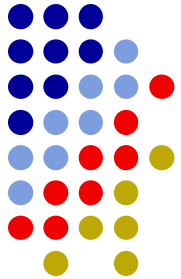
Zip-state (zip, state)

We've lost the FD *number street city state -> zip*

If we put this table back in the design, we are back where we started. And we violate BCNF.

# Algorithm for lossless join decomposition into BCNF relations

**(not necessarily dependency preserving)**



1. set D := { R } (the current set of relations)

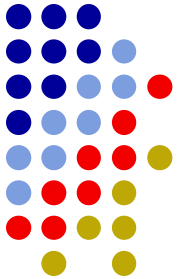
2. while there is a relation in R that is not in BCNF
   begin
      choose a relation Q that is not in BCNF
      find an FD X → Y in Q that violates BCNF
      replace Q in D by two relations: (Q - Y) and (X U Y)
   end;

Lifting "troublesome" FD

Finding a "troublesome" FD

# Algorithm for dep. preserving, lossless join decomposition into ~~BCNF~~ 3NF relations

**(another example of available algorithms)**



Same as before

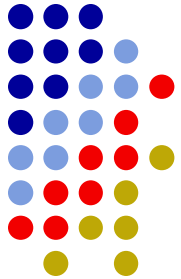
1. set D := { R } (the current set of relations)

2. while there is a relation in R that is not in BCNF
   begin
   choose a relation Q that is not in BCNF
   find an FD X → Y in Q that violates BCNF
   replace Q in D by two relations: (Q - Y) and (X U Y)
   end;

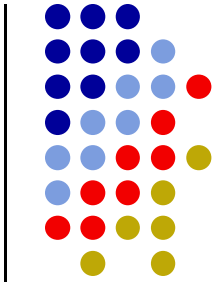
dep preserving

3. identify dependencies that are not preserved (X → A).
   add XA as a table to the set D

# There are a number of other results:

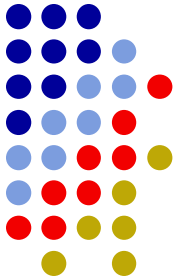


- algorithm to compute F+
- algorithm to find a minimal cover for a set of FDs
- algorithm for dependency-preserving decomposition into 3NF
- algorithm to synthesize tables, given a set of attributes and a set of FDs



# A few comments

# Sometimes redundancy has NOTHING to do with FDs



Suppose we have two tables for employee information.

Employee(ssn, name, salary, birthdate)
Employee2(ssn, name, home-address)

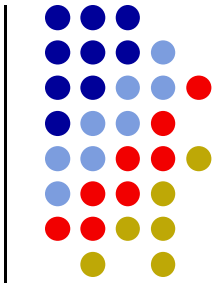
And, we put the name in both tables, for convenience.

Name is stored redundantly.

And, it is possible for the two names to be inconsistent, if you change it in one place but not in another.

This redundancy is NOT caused by a troublesome FD

# Sometimes redundancy has NOTHING to do with FDs



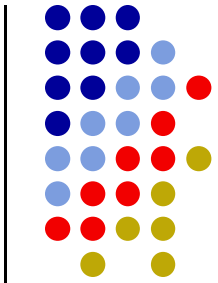
Foreign keys are necessarily … by definition … introducing redundancy.

Student(id, name, major, advisor-num)

Faculty(id, name, rank)

The faculty.id value is repeated in the Student.advisor-num attribute for every student that has this faculty member as an advisor.

# Null Values are Useful



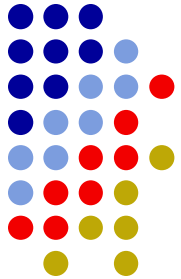
Makes the DB more flexible; makes it simpler to insert data, for example

For example:

Employee(ssn, name, DOB, spouse)

We might not know the DOB (when data is entered) or this employee might not have a spouse.

# Null values cause problems

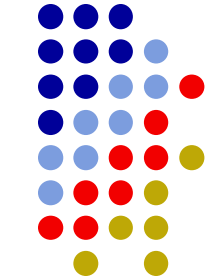


- may waste space

- may have different meanings:
  - attribute *does not apply* to this tuple
  - attribute value is *unknown*
  - value is *known but absent* (not yet recorded)

- may make queries harder to write
  - need to use outer joins (rather than joins)
  - may make aggregates harder to understand

# Null Values Cause Problems for Aggregate Operators



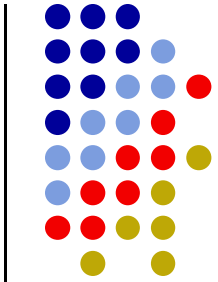
Employee (ssn, name, salary)

SELECT AVG(salary) FROM Employee;

SELECT SUM(salary) INTO salsum FROM Employee;
SELECT COUNT(*) INTO total FROM Employee;

Salsum/total might be different from first query answer. How could that happen?

# Decomposition Reduces the Use of Null Values

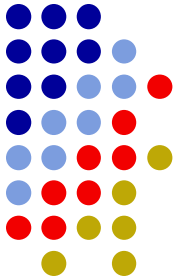


Use two tables:
Employee (ssn, name, DOB)
Employee-extra (ssn, spouse)

Rather than:
Employee(ssn, name, DOB, spouse)

Generally, it is better to reduce the use of null values, if you can. The first design, above, doesn't require the use of null values for spouse.

# Normalization (decomposition) reduces the use of null values



Employee(SS-number, name, project, p-title)

decomposition: Employee (SS-number, name)
Project (p-num, p-title, name)

If an employee hasn't been assigned a project yet, the normalized schema doesn't require ANY null values.

# A few details … using Armstrong's axioms

Supplement to Normalization Lecture

Lois Delcambre

# Armstrong's Axioms – with explanation and examples

Reflexivity: If $X \supseteq Y$, then $X \rightarrow Y$. (identity function is a function)

Augmentation: If $X \rightarrow Y$, then $XZ \rightarrow YZ$, for any Z. (parallel application of one function and the identity function is a function)

Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$. (composition of two functions is a function)

Examples:

Reflexivity: ssn→ssn, ssn,name→ssn

Augmentation: If ssn→name then ssn,color→name,color

Transitivity: If ssn→mgr-id and mgr-id→mgr-name, then ssn→mgr-name.

# Using Armstrong's Axioms

Reflexivity: If $X \supseteq Y$, then $X \rightarrow Y$.

Augmentation: If $X \rightarrow Y$, then $XZ \rightarrow YZ$, for any Z.

Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.

Decomposition: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$

Proof:

| | |
|---|---|
| $X \rightarrow YZ$ | given |
| $YZ \rightarrow Y$ | Reflexivity (trivial FD) |
| $X \rightarrow Y$ | Transitivity |

(Similarly, $X \rightarrow Z$.)

# Using Armstrong's Axioms (cont.)

Proposition - Superkeys can be derived from keys:

If $X \rightarrow Y$, then $XA \rightarrow Y$, for any A

Example: If SSN $\rightarrow$ name, then SSN,color $\rightarrow$ name

Proof:

| | |
|---|---|
| $X \rightarrow Y$ | Given |
| $XA \rightarrow YA$ | Augmentation |
| $XA \rightarrow Y$ | Decomposition (proved on previous slide) |

I can construct a superkey from a key.

# Using Armstrong's Axioms (cont.)

Proposition: If $X \rightarrow A$ and X is a superkey (and not a key) for the table, then $X \rightarrow A$ is derivable from a key.

Proof:

| | |
|---|---|
| $X \rightarrow A$ | Given |
| $X = Y \cup Z$ where: | |
| Y is a key, | |
| Z is non-empty, | |
| Y and Z disjoint | Because X is a superkey but not a key |
| $Y \rightarrow A$ | Because Y is a key for the table that A is in |

Therefore $X \rightarrow A$ follows from: $Y \rightarrow A$ (implied by the key) and the result from the previous slide.

# Formal definition of BCNF
## (in the textbook) - revisited

- For a table R, every FD X → A that occurs among attributes of R then either:
  - A is an element of X (X → A is trivial)
  - ~~A is part of a key (don't worry about "key" attributes)~~
  - X is a superkey of R
    consider the following 2 cases:
    - X is a key for R (good)
    - X is a superkey for R (and not a key). The X → A is derivable from a key using augmentation and decomposition.

For a table to be in BCNF, every FD is either trivial or derivable from the FDs implied by the key(s).

Informally, I often say, BCNF if all FDs are implied by the key(s).

# Formal definition of 3NF
## (in the textbook)

- For a table R, every FD X → A that occurs among attributes of R then either:
  - A is an element of X (X → A is trivial)
  - A is part of a key (ignore the "key" attributes)
  - X is a superkey of R
    Consider the following 2 cases:
    - X is a key for R (good)
    - X is a superkey for R (and not a key). The X → A is derivable from a key using augmentation. (Stay tuned.)

A table is in 3NF if all the non-key attributes are either trivial or implied by FDs derivable from the FDs implied by the key(s).

# Using Armstong's Axioms to show dependency preservation (that SSN →dname is not lost)

Employee (SSN, name, phone, dept, dept-name)

Employee (SSN, name, phone, dept)

Department (dept, dname)

F = {SSN→name, SSN→phone, SSN→dept, SSN→dept-name, dept→dname} original set of FDs

G = {SSN→name, SSN→phone, SSN→dept, ~~SSN →dept-name~~, dept→dname} the set of FDs projected from F

But $G^+$ includes SSN→dept-name because we can derive it:

| | |
|---|---|
| SSN→dept | Given (it is in G) |
| dept→dname | Given (it is in G) |
| SSN→dname | Because of transitivity. |

# Example showing that we must project from $F^+$ when considering dependency preservation

R(<u>a, b</u>, c) where ab is a key, a $\rightarrow$b, b$\rightarrow$a, a $\rightarrow$c

F = {ab$\rightarrow$c, b$\rightarrow$a, a$\rightarrow$b, a$\rightarrow$c}

Suppose we decompose to
X(a, b) and Y(b, c)

If we project F onto X and Y, we see:
a$\rightarrow$b, b$\rightarrow$a and that's it. We appear to have lost many FDs.

But $F^+$ includes this additional FD:
b$\rightarrow$c (because b$\rightarrow$a and a$\rightarrow$c)

If we project $F^+$ onto X and Y we see:
a$\rightarrow$b, b$\rightarrow$a , and b$\rightarrow$c in G.

$G^+$ then includes a$\rightarrow$b, b$\rightarrow$a , b$\rightarrow$c, plus a$\rightarrow$c (by transitivity), ab$\rightarrow$c (by augmentation).

# Same example using realistic attribute names

R(ssn, id, name)

where (ssn, id) is a key,

F = {ssn →id, id→ssn, ssn→name}

X(ssn, id)  Y(id, name)

Projection of F = {ssn →id, id→ssn}

But $F^+$ includes id→name (because id →ssn, and ssn →name)

Projection of $F^+$ includes {ssn →id, id→ssn, id →name}

From that projection, we can compute G+ which includes
ssn→name (because ssn→id, id→name)

# Challenge Question

Proposition:

If $AB \rightarrow C$ (where A and B are disjoint sets of attributes)

then $A \rightarrow C$ and $B \rightarrow C$.

Is this true or false?

Can you prove/disprove it?

# Other Normalization Results

- When a table is in BCNF, it is not possible to have redundancies or update anomalies caused by FDs.
- There are other dependencies besides FDs.
  - Multi-valued dependency … leads to the definition of 4NF
  - Join dependency … leads to the definition of 5NF

- For FDs, MVDs, and JDs, the project operator is used to decompose and the join operator is used to recontruct.
- There are no redundancies or update anomalies that remain in a table in 5NF that can be solved by projecting/joining. 5NF is sometimes PJNF.

# Normalization made easy

Every attribute in a table must depend on the
key (definition of a key),
the whole key (2NF – no partial dependencies)
and
nothing but the key
(3NF – no transitive dependencies).

Every non-key attribute in a table must depend on the
key (definition of a key)
the whole key (2NF – no partial dependencies)
and
nothing but the key
(3NF – no transitive dependencies).